1. **Plan (Bold parts involve logic reasoning based on the KB of the agent):**
    1.1. Perceive current cell
    1.2. Ask KB for safe rooms
    1.3. If KB tells that GLITTER is perceived, then grab gold and, go to a safe place and climb out.
    1.4. Ask KB for unvisited rooms
    1.5. Plan a route from current pos to a safe unvisited state (not bold since we have already used logic reasoning to find safe states, and finding unvisited states is trivial)
    1.6. If we have the arrow, shoot at most likely Wumpus location
    1.7. If at this point we don't have any safe actions, we need to take a risk and go to an unvisited notUnsafe room.
    1.8. Go home

2. **Prover called:** The number of calls is (CaveXDimension * CaveYDimension) - (size of Set<Room> visited), with argument OK_t_x_y, so once for all unvisited rooms. HaveArrow_t is called if the we run out of safe states to visit.
   Queries: "ASK(KB, OK^t_{x,y})"
3. It's O(2^n) since worst case scenario we have to check all symbols.
4. **The first queries and their corresponding time in ms are:**
    Query: ~OK_0_1_2 Time: 196ms
    Query: ~OK_0_1_3 Time: 150ms
    Query: ~OK_0_2_1 Time: 180ms
    Query: ~OK_0_2_2 Time: 108ms
    Query: ~OK_0_2_3 Time: 133ms
    Query: ~OK_0_3_1 Time: 128ms
    Query: ~OK_0_3_2 Time: 145ms
    Query: ~OK_0_3_3 Time: 149ms
    Query: ~OK_1_1_3 Time: 243ms

   For ~OK_0_1_2 the number of calls to dpll were 344874. Main propositional symbols are: W, P, B, S and OK for each squares, in total we have 9 * 5 = 45 propositional symbols (omitting others, like HaveArrow etc). According to questions 3 then the upper limit should be 2*45 ~ 3e13, so we are far from the theoretical limit.

5.
   For 3x3 board the first queries and their corresponding time in ms are:
    Query: ~OK_0_1_2 Time: 11ms
    Query: ~OK_0_1_3 Time: 6ms
    Query: ~OK_0_2_1 Time: 8ms
    Query: ~OK_0_2_2 Time: 5ms
    Query: ~OK_0_2_3 Time: 5ms
    Query: ~OK_0_3_1 Time: 4ms
    Query: ~OK_0_3_2 Time: 3ms
    Query: ~OK_0_3_3 Time: 15ms
    Query: ~OK_1_1_3 Time: 6ms

For 5x5 our algorithm does not work without the heuristics. Works fine with heuristics.

6. The book mentions a few tricks that help SAT solvers with larger problems. Two of them are:
    - **Variable and value ordering.** Our implementation uses arbitrary ordering, but we could instead try the most frequent variables for better results.
    - **Random restarts:** We can get stuck going down a bad branch, so randomly restarting will guide the search down to another branch due to the random nature of our algorithm (of variable selection) and possibly get us closer to a solution.
7. Added **Random restarts**. The impact was minimal. Time for first move on a 3x3 table was 181ms without and 177ms with random restarts. This difference is not statisticaly significant. Note that this was done on a different computer than before so the results are not comparable with previous questions.

Code from question 7 follows.
You can also download here:
https://www.dropbox.com/s/iyk8v6u21s03pun/DPLL.java.pdf?dl=0

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

import java.util.concurrent.ThreadLocalRandom;

import aima.core.logic.propositional.inference.InferenceProcedure;
import aima.core.logic.propositional.kb.KnowledgeBase;
import aima.core.logic.propositional.kb.data.Clause;
import aima.core.logic.propositional.kb.data.Literal;
import aima.core.logic.propositional.kb.data.Model;
import aima.core.logic.propositional.parsing.ast.ComplexSentence;
import aima.core.logic.propositional.parsing.ast.Connective;
import aima.core.logic.propositional.parsing.ast.PropositionSymbol;
import aima.core.logic.propositional.parsing.ast.Sentence;
import aima.core.logic.propositional.visitors.ConvertToConjunctionOfClauses;
import aima.core.logic.propositional.visitors.SymbolCollector;
import aima.core.util.Tasks;
import aima.core.util.Util;
import aima.core.util.datastructure.Pair;

public class DPLL implements InferenceProcedure {
```

```
	private int randomRestartMax = 500000000;

	/**
	 * Determine if KB |= &alpha;, i.e. alpha is entailed by KB.
	 *
	 * @param kb
	 *          a Knowledge Base in propositional logic.
	 * @param alpha
	 *          a propositional sentence.
	 * @return true, if &alpha; is entailed by KB, false otherwise.
	 */
	@Override
	public boolean isEntailed(KnowledgeBase kb, Sentence alpha) {
		// AIMA3e p.g. 260: kb |= alpha, can be done by testing
		// unsatisfiability of kb & ~alpha.
		Set<Clause> kbAndNotAlpha = new LinkedHashSet<>();
		Sentence notQuery = new ComplexSentence(Connective.NOT, alpha);
		Set<PropositionSymbol> symbols = new LinkedHashSet<>();
		List<PropositionSymbol> querySymbols = new
ArrayList<>(SymbolCollector.getSymbolsFrom(notQuery));
		long tStart = System.currentTimeMillis();

		kbAndNotAlpha.addAll(kb.asCNF());

kbAndNotAlpha.addAll(ConvertToConjunctionOfClauses.convert(notQuery).getClauses());
		symbols.addAll(querySymbols);
		symbols.addAll(kb.getSymbols());



		boolean dpllValue;
		while (true) {
			try {
				dpllValue = dpll(kbAndNotAlpha, new ArrayList<>(symbols),
new Model());
				long runTime = System.currentTimeMillis() - tStart;
				System.out.print(notQuery+" | "+runTime+" ms;  ");
				return dpllValue;
			} catch (RestartException e) {
//				System.out.println("Restarting Search");
			}
		}
	}

	/**
	 * DPLL-SATISFIABLE?(s)<br>
```

```
	 * Checks the satisfiability of a sentence in propositional logic.
	 * 
	 * @param s
	 *            a sentence in propositional logic.
	 * @return true if the sentence is satisfiable, false otherwise.
	 */
	public boolean dpllSatisfiable(Sentence s) {
		// clauses <- the set of clauses in the CNF representation of s
		Set<Clause> clauses = 
ConvertToConjunctionOfClauses.convert(s).getClauses();
		// symbols <- a list of the proposition symbols in s
		List<PropositionSymbol> symbols = getPropositionSymbolsInSentence(s);

		// return DPLL(clauses, symbols, {})
		return dpll(clauses, symbols, new Model());
	}

	/**
	 * DPLL(clauses, symbols, model)<br>
	 * 
	 * @param clauses
	 *            the set of clauses.
	 * @param symbols
	 *            a list of unassigned symbols.
	 * @param model
	 *            contains the values for assigned symbols.
	 * @return true if the model is satisfiable under current assignments, false
	 *         otherwise.
	 */
	public boolean dpll(Set<Clause> clauses, List<PropositionSymbol> symbols, Model 
model) throws RestartException{
		// if every clause in clauses is true in model then return true
		// if some clause in clauses is false in model then return false
		// NOTE: for optimization reasons we only want to determine the
		// values of clauses once on each call to dpll
		int randomNum = ThreadLocalRandom.current().nextInt(0, 
randomRestartMax + 1);
		if (randomNum == 0)
			throw new RestartException();

		boolean allTrue = true;
		Set<Clause> unknownClauses = new LinkedHashSet<>();
		for (Clause c : clauses) {
			Boolean value = model.determineValue(c);
			if (!Boolean.TRUE.equals(value)) {
				allTrue = false;
```

```
                        if (Boolean.FALSE.equals(value)) {
                            return false;
                        }
                        unknownClauses.add(c);
                    }
                }
                if (allTrue) {
                    return true;
                } else if (Tasks.currIsCancelled())
                    return false;

                // NOTE: Performance Optimization -
                // Going forward, algorithm can ignore clauses that are already
                // known to be true (reduces overhead on recursive calls).
                clauses = unknownClauses;

                // TODO: add remaining parts of PDDL algorithm here

        /*      // P, value←FIND-PURE-SYMBOL(symbols, clauses,model )
                Pair<PropositionSymbol, Boolean> pure = findPureSymbol(symbols, clauses,
model);

                if (pure != null) {
                    symbols.remove(pure.getFirst());
                    return callDPLL(clauses, symbols, model, pure.getFirst(),
pure.getSecond());
                }

                // P, value←FIND-UNIT-CLAUSE(clauses,model )
                Pair<PropositionSymbol, Boolean> unit = findUnitClause(clauses, model);

                if (unit != null) {
                    symbols.remove(unit.getFirst());
                    return callDPLL(clauses, symbols, model, unit.getFirst(),
unit.getSecond());
                }
*/
                // P <- FIRST(symbols); rest <- REST(symbols)
                PropositionSymbol p = Util.first(symbols);
                List<PropositionSymbol> rest = Util.rest(symbols);
                // return DPLL(clauses, rest, model U {P = true}) or
                // ...... DPLL(clauses, rest, model U {P = false})
                return callDPLL(clauses, rest, model, p, true) || callDPLL(clauses, rest, model,
p, false);
        }
```

```
	// END-DPLL
	//

	//
	// PROTECTED
	//

	private boolean callDPLL(Set<Clause> clauses, List<PropositionSymbol> symbols,
Model model, PropositionSymbol p,
			boolean value) throws RestartException{
		// We update the model in place with the assignment p=value,
		boolean result = dpll(clauses, symbols, model.unionInPlace(p, value));
		// as backtracking can occur during the recursive calls we
		// need to remove the assigned value before we pop back out from this
		// call.
		model.remove(p);
		return result;
	}

	// Note: Override this method if you wish to change the initial variable
	// ordering when dpllSatisfiable is called.
	protected List<PropositionSymbol> getPropositionSymbolsInSentence(Sentence s) {
		List<PropositionSymbol> result = new
ArrayList<PropositionSymbol>(SymbolCollector.getSymbolsFrom(s));

		return result;
	}

	/**
	 * AIMA3e p.g. 260:<br>
	 * <quote><i>Pure symbol heuristic:</i> A <b>pure symbol</b> is a symbol that
	 * always appears with the same "sign" in all clauses. For example, in the three
	 * clauses (A | ~B), (~B | ~C), and (C | A), the symbol A is pure because only
	 * the positive literal appears, B is pure because only the negative literal
	 * appears, and C is impure. It is easy to see that if a sentence has a model,
	 * then it has a model with the pure symbols assigned so as to make their
	 * literals true, because doing so can never make a clause false. Note that, in
	 * determining the purity of a symbol, the algorithm can ignore clauses that are
	 * already known to be true in the model constructed so far. For example, if the
	 * model contains B=false, then the clause (~B | ~C) is already true, and in the
	 * remaining clauses C appears only as a positive literal; therefore C becomes
	 * pure.</quote>
	 *
	 * @param symbols
	 *	a list of currently unassigned symbols in the model (to be checked
	 *	if pure or not).
```

```
	 * @param clauses
	 * @param model
	 * @return a proposition symbol and value pair identifying a pure symbol and a
	 *         value to be assigned to it, otherwise null if no pure symbol can be
	 *         identified.
	 */
	protected Pair<PropositionSymbol, Boolean>
findPureSymbol(List<PropositionSymbol> symbols, Set<Clause> clauses,
			Model model) {
		Pair<PropositionSymbol, Boolean> result = null;

		Set<PropositionSymbol> symbolsToKeep = new HashSet<>(symbols);
		// Collect up possible positive and negative candidate sets of pure
		// symbols
		Set<PropositionSymbol> candidatePurePositiveSymbols = new HashSet<>();
		Set<PropositionSymbol> candidatePureNegativeSymbols = new
HashSet<>();
		for (Clause c : clauses) {
			// Algorithm can ignore clauses that are already known to be true
			// NOTE: no longer need to do this here as we remove, true clauses
			// up front in the dpll call (as an optimization)

			// Collect possible candidates, removing all candidates that are
			// not part of the input list of symbols to be considered.
			for (PropositionSymbol p : c.getPositiveSymbols()) {
				if (symbolsToKeep.contains(p)) {
					candidatePurePositiveSymbols.add(p);
				}
			}
			for (PropositionSymbol n : c.getNegativeSymbols()) {
				if (symbolsToKeep.contains(n)) {
					candidatePureNegativeSymbols.add(n);
				}
			}
		}

		// Determine the overlap/intersection between the positive and negative
		// candidates
		for (PropositionSymbol s : symbolsToKeep) {
			// Remove the non-pure symbols
			if (candidatePurePositiveSymbols.contains(s) &&
candidatePureNegativeSymbols.contains(s)) {
				candidatePurePositiveSymbols.remove(s);
				candidatePureNegativeSymbols.remove(s);
			}
		}
```

```
		// We have an implicit preference for positive pure symbols
		if (candidatePurePositiveSymbols.size() > 0) {
			result = new Pair<>(candidatePurePositiveSymbols.iterator().next(),
true);
		} // We have a negative pure symbol
		else if (candidatePureNegativeSymbols.size() > 0) {
			result = new Pair<PropositionSymbol,
Boolean>(candidatePureNegativeSymbols.iterator().next(), false);
		}

		return result;
	}

	/**
	 * AIMA3e p.g. 260:<br>
	 * <quote><i>Unit clause heuristic:</i> A <b>unit clause</b> was defined earlier
	 * as a clause with just one literal. In the context of DPLL, it also means
	 * clauses in which all literals but one are already assigned false by the
	 * model. For example, if the model contains B = true, then (~B | ~C) simplifies
	 * to ~C, which is a unit clause. Obviously, for this clause to be true, C must
	 * be set to false. The unit clause heuristic assigns all such symbols before
	 * branching on the remainder. One important consequence of the heuristic is
	 * that any attempt to prove (by refutation) a literal that is already in the
	 * knowledge base will succeed immediately. Notice also that assigning one unit
	 * clause can create another unit clause - for example, when C is set to false,
	 * (C | A) becomes a unit clause, causing true to be assigned to A. This
	 * "cascade" of forced assignments is called <b>unit propagation</b>. It
	 * resembles the process of forward chaining with definite clauses, and indeed,
	 * if the CNF expression contains only definite clauses then DPLL essentially
	 * replicates forward chaining.</quote>
	 *
	 * @param clauses
	 * @param model
	 * @return a proposition symbol and value pair identifying a unit clause and a
	 *        value to be assigned to it, otherwise null if no unit clause can be
	 *        identified.
	 */
	protected Pair<PropositionSymbol, Boolean> findUnitClause(Set<Clause> clauses,
Model model) {
		Pair<PropositionSymbol, Boolean> result = null;

		for (Clause c : clauses) {
			// if clauses value is currently unknown
			// (i.e. means known literals are false)
			// NOTE: no longer need to perform this check
```

```
                        // as only clauses with unknown values will
                        // be passed to this routine from dpll as it
                        // removes known ones up front.
                        Literal unassigned = null;
                        // Default definition of a unit clause is a clause
                        // with just one literal
                        if (c.isUnitClause()) {
                                unassigned = c.getLiterals().iterator().next();
                        } else {
                                // Also, a unit clause in the context of DPLL, also means a
                                // clauseF in which all literals but one are already
                                // assigned false by the model.
                                // Note: at this point we already know the clause is not
                                // true, so just need to determine if the clause has a
                                // single unassigned literal
                                for (Literal l : c.getLiterals()) {
                                        Boolean value =
model.getValue(l.getAtomicSentence());
                                        if (value == null) {
                                                // The first unassigned literal encountered.
                                                if (unassigned == null) {
                                                        unassigned = l;
                                                } else {
                                                        // This means we have more than 1
unassigned
                                                        // literal so lets skip
                                                        unassigned = null;
                                                        break;
                                                }
                                        }
                                }
                        }

                        // if a value assigned it means we have a single
                        // unassigned literal and all the assigned literals
                        // are not true under the current model as we were
                        // unable to determine a value.
                        if (unassigned != null) {
                                result = new Pair<>(unassigned.getAtomicSentence(),
unassigned.isPositiveLiteral());
                                break;
                        }
                }

                return result;
        }
```

```
	// symbols - P
	protected List<PropositionSymbol> minus(List<PropositionSymbol> symbols,
PropositionSymbol p) {
		List<PropositionSymbol> result = new ArrayList<>(symbols.size());
		for (PropositionSymbol s : symbols) {
			// symbols - P
			if (!p.equals(s))
				result.add(s);
		}
		return result;
	}
}
```